ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to Avalon-MM Interface

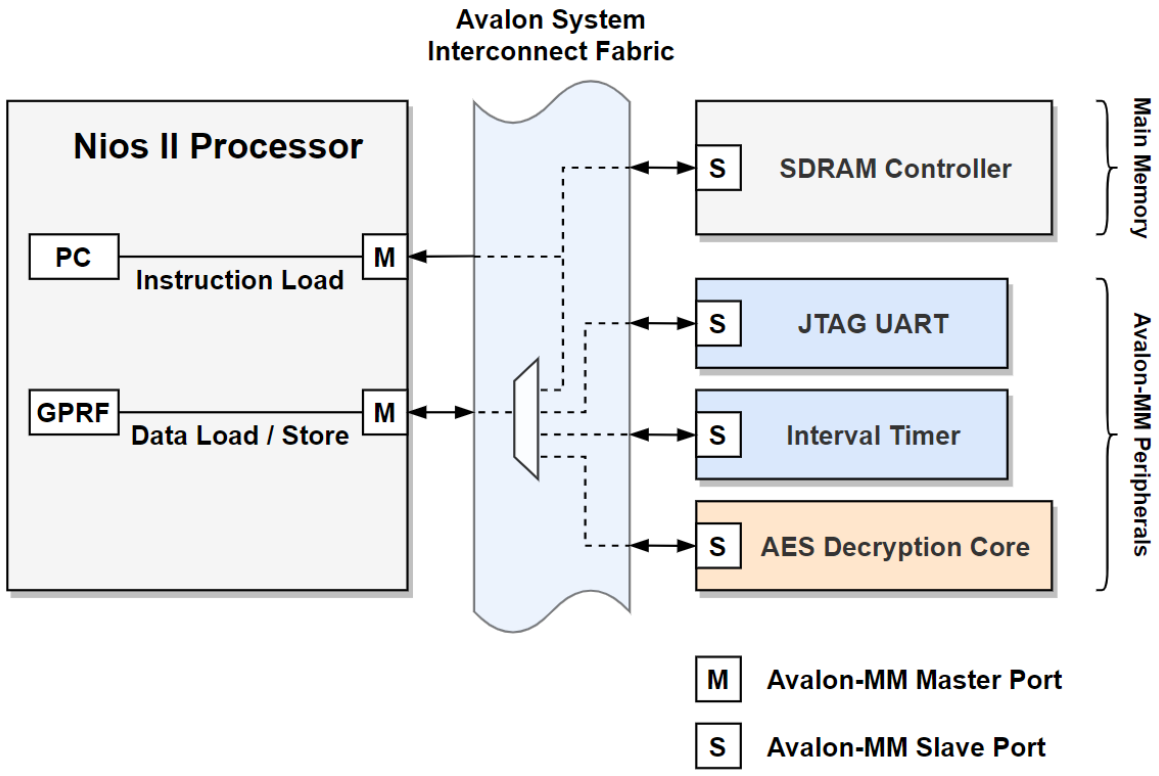
Please read this guide carefully and thoroughly as minor mistakes can impact the functionality of your entire project.

System Overview

In this lab, you will create an AES decryption IP core and interface it with the Avalon-MM (memory mapped) interconnect. Nios II will perform encryption in software and communicate with your IP to perform decryption in hardware. Your IP core will have an Avalon-MM slave port that allows Avalon masters such as Nios II to directly access it with read/write operations.

To perform decryption, Nios II will write the 128-bit encrypted message and key to the AES decryption core, then write a start signal to one of its registers. We use polling to wait until decryption is complete in hardware and finally read back the 128-bit decrypted message.

Figure 1. High-level System Block Diagram



Note: not all peripherals are shown in this figure. (PLLs, System ID, etc.)

To begin, start with your Platform Designer setup from Lab 6.2 with the usual components like Nios II, SDRAM, and JTAG UART, and remove the unnecessary USB components. Save and close it for now. In the next section, you will create your own AES decryption core component and add it to Platform Designer.

The AES Decryption Core Interface

The interface module *avalon_aes_interface.sv* will be the top-level file for the AES decryption core component on Platform Designer (note that *lab7_top.sv* is still the top-level file for the entire project). We have provided the input/output signals declaration for you. There is a clock input (CLK), an active-high reset input (RESET), an exported conduit signal which is just a 32-bit output (EXPORT_DATA), and finally an Avalon-MM slave port which contains a bundle of signals whose specifications are given below.

The Avalon-MM slave port will complete read and write operations requested by its master, the Nios II processor (read/write operations correspond to load/store instructions in Nios). While the Avalon specifications provide many signals to use for its interface, we only need to use 7 signals to implement the slave port for this lab.

Table 1. Avalon-MM Slave Port Interface Signals

Name	Direction	Width	Description
read	Input	**1**	High when a read operation is to be performed.
write	Input	**1**	High when a write operation is to be performed.
readdata	Output	**32**	32-bit data to be read.
writedata	Input	**32**	32-bit data to be written.
address	Input	**4**	Address of the read or write operation.
byteenable	Input	**4**	4-bit active high signal to identify which byte(s) are being written.
chipselect	Input	**1**	High during a read or write operation.

Note that the data width of 32-bit and address width of 4-bit are chosen for this lab, they may be up to 1024-bit and 64-bit, respectively. We are using 32-bit **readdata** and **writedata** signals because they match the data width of Nios II, a 32-bit processor. As for the 4-bit address, which gives $2^4 = 16$ locations, this is the minimum needed to store the encrypted message, key, decrypted message, and control signals as we will discuss in the next section.

Now it is up to you to implement the body of this module that completes the incoming read and write requests. Internally, you should create 16 registers, each 32-bit, that hold the values being read and written. There are some requirements that your design must satisfy:

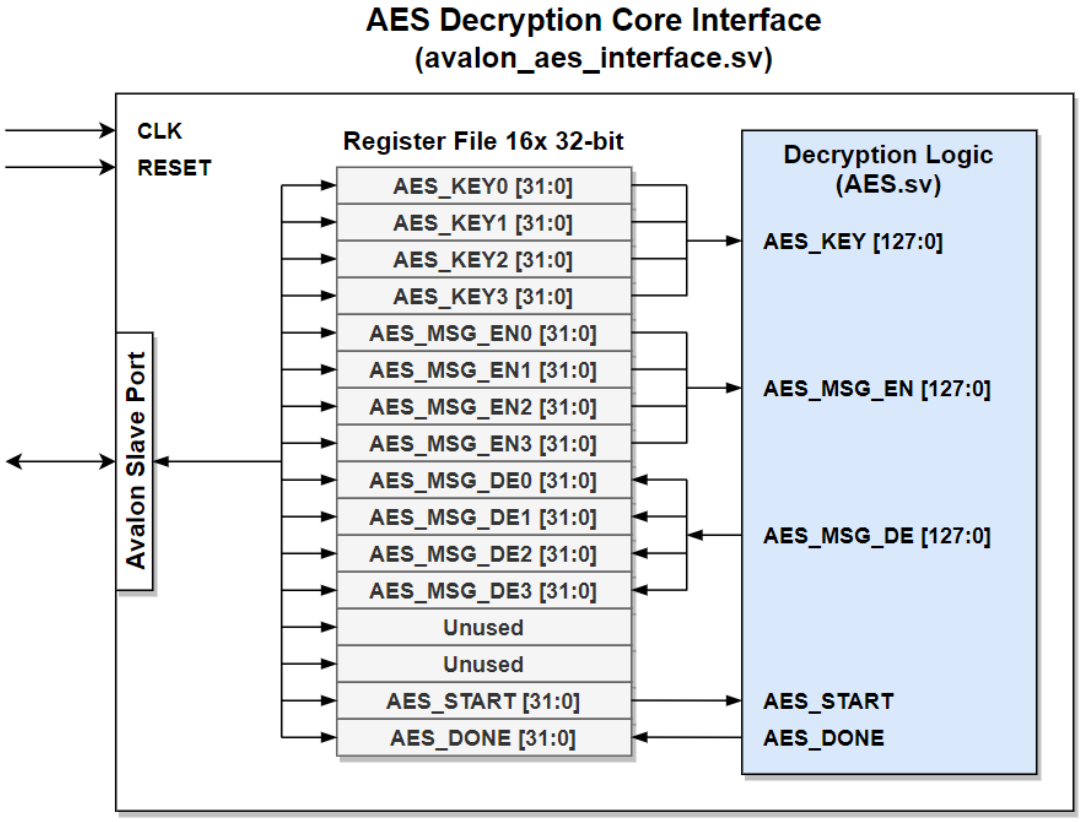
- Read has a 0 cycle wait latency. This means that when read is high, **readdata** should have the value of the addressed register on the same cycle.
- Write has a 0 cycle wait latency. This means that when write is high, the addressed register should be updated with **writedata** on the very next cycle.
- Byte enable determines the bytes being written according to the table below.

Table 2. Byte Enable Description

byteenable[3:0]	Write Action
1111	Write full 32-bits.
1100	Write the two upper bytes.
0011	Write the two lower bytes.
1000	Write byte 3 only.
0100	Write byte 2 only.
0010	Write byte 1 only.
0001	Write byte 0 only.

Nios II will send the 128-bit AES key and encrypted message as 4 writes ($4 \times 32\ bit = 128\ bit$), thus we need 4 registers to hold the AES key, encrypted message, and decrypted message, in total 12 registers each 32-bit. We need 2 more registers to hold the START and DONE signals that Nios II will use to control the hardware state, this brings the total to 14. Since address ranges must be powers of 2, we will use 14 of the 16 addressable registers, the remaining two are unused. The recommended implementation of *avalon_aes_interface.sv* is shown in Figure 2.

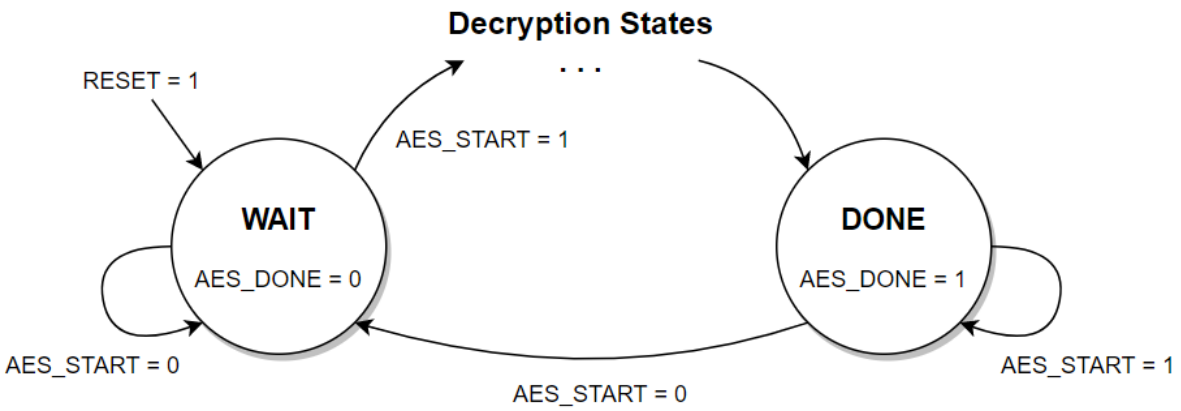
Figure 2. AES Decryption Core Interface



For week 1, implementing the register array and making sure that you can write your AES key and encrypted message to them will suffice. Keep in mind that EXPORT_DATA should be assigned the first 2 and last bytes of the AES key, exported from the Platform Designer design (covered in next section), and displayed on the LEDs with HEX drivers on the lab 7 top level.

For week 2, instantiate *AES.sv*, the decryption logic with your state machine, and make the appropriate connections by packing or unpacking the 4x 32-bit registers to 128-bit input and output ports for AES key, encrypted message, and decrypted message. The AES_START and AES_DONE signals are 1 bit so you can simply use the least significant bits of registers 14 and 15. Your state machine in *AES.sv* should be able to perform decryption continuously. Create two control states: WAIT and DONE to handle the AES_START input and AES_DONE output signals as shown in Figure 3.

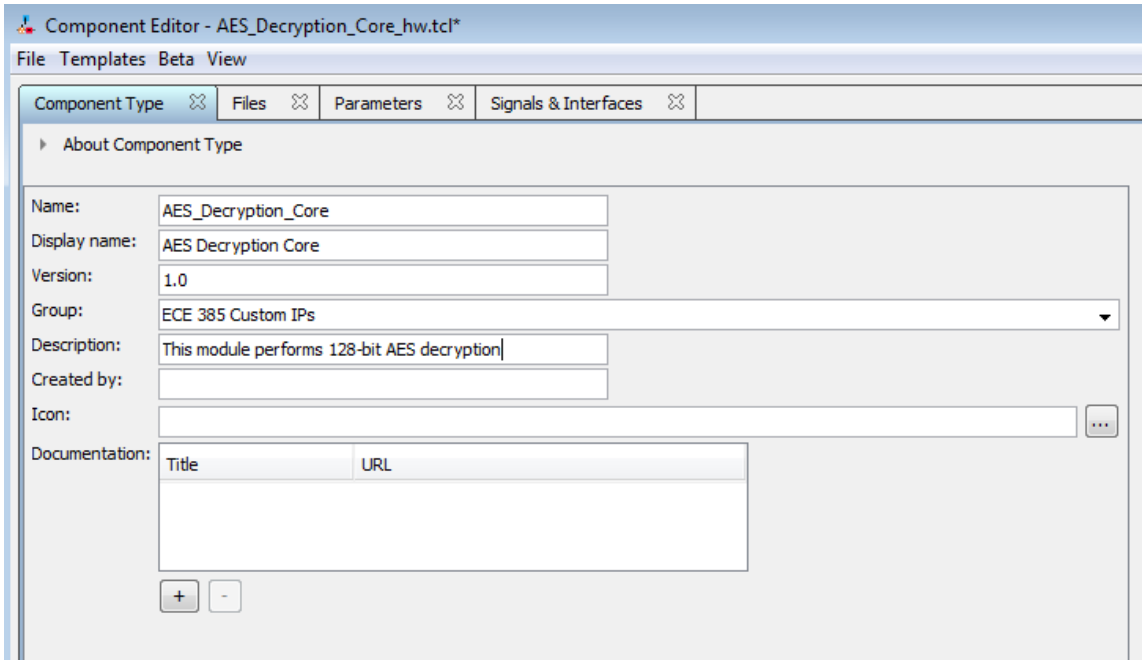
Figure 3. Control States of the Decryption FSM



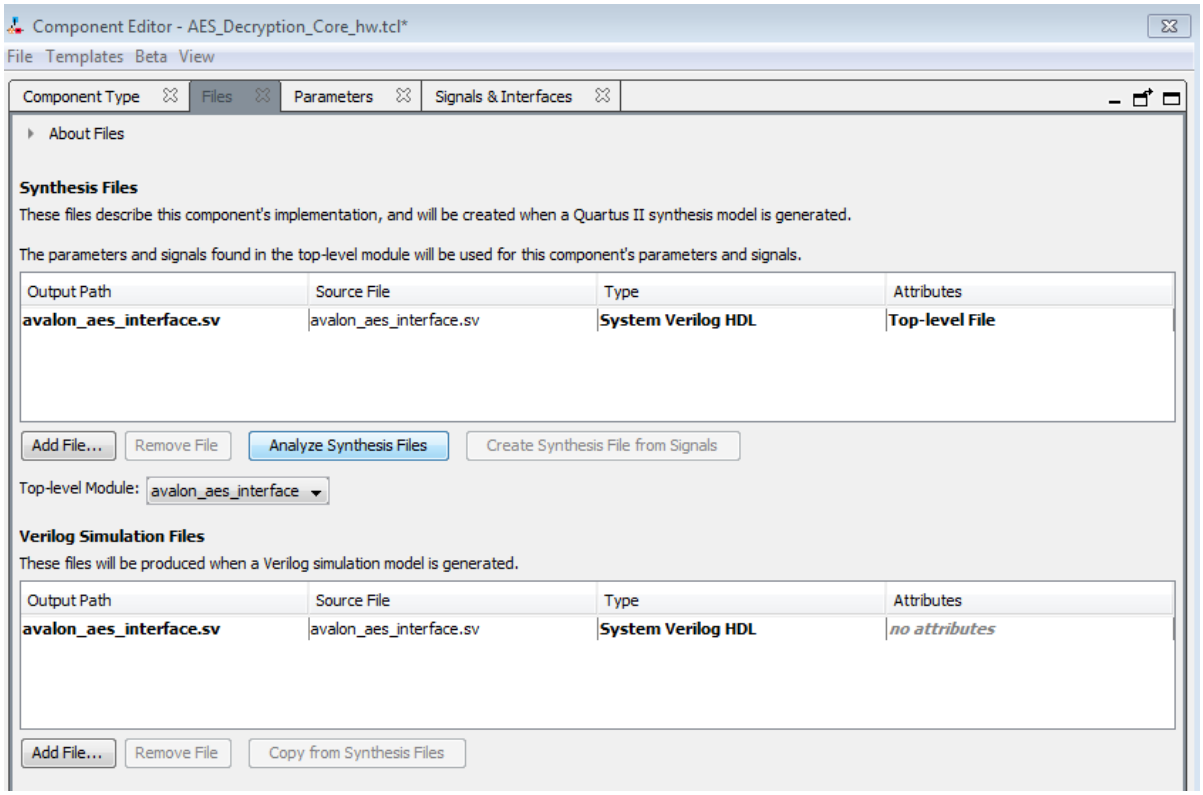
Creating a Platform Designer Component

When you are done implementing the partial (week 1) or complete interface module *avalon_aes_interface.sv*, follow these steps to add it to the Platform Designer IP catalog with component editor.

1. Launch Platform Designer editor and load your design that already has Nios II, SDRAM, UART, etc.
2. On the upper left **IP Catalog** panel, double click ***New Component…***

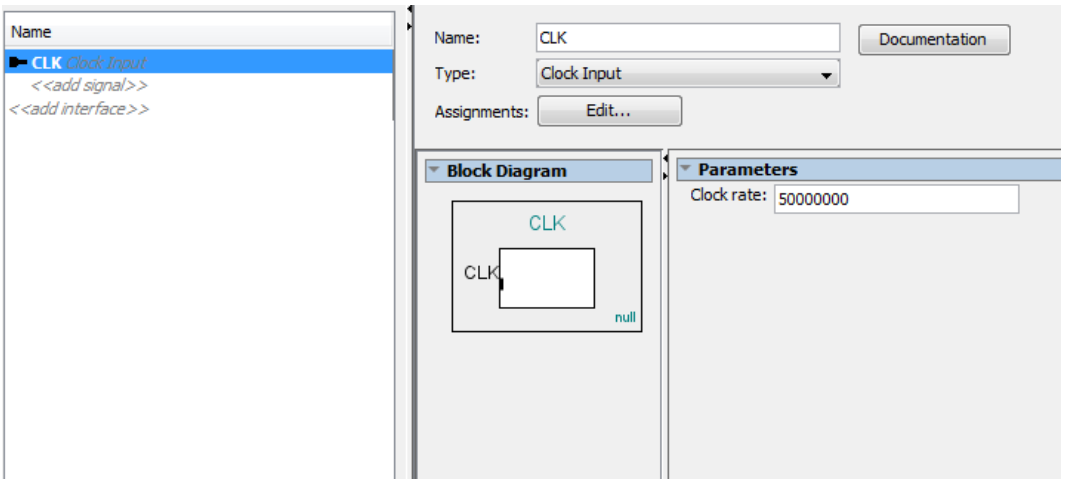


3. Enter the Name and Display name for your component. Display name is the one shown in Platform Designer IP Catalog. It is good practice to also keep track of the version of your IP, increment it when you make changes or fix issues. You can also categorize it to an existing or new group, here we make a group called “ECE 385 Custom IPs”. Finally, give it a brief description.
4. Click on the Files tab.



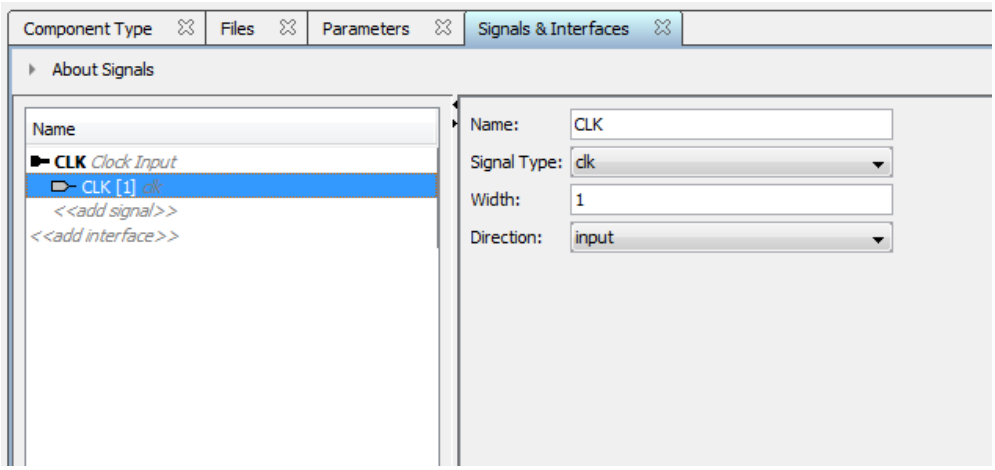
5. Under **Synthesis Files**, click on Add File… and choose *avalon_aes_interface.sv* which is the top-level module for this component. Under **Verilog Simulation Files**, click on **Copy from Synthesis Files**, this is useful if you want to simulate your system in ModelSim.
6. Click on the Signals & Interfaces tab. Now we want to create the Avalon ports and match the Avalon defined interface signals with our input/output declarations in *avalon_aes_interface.sv*.
7. Let us add the clock input first. Click on *<<add interface>>*.
   - In Quartus 15.0, a default name of “avalon_slave” will appear, replace that with “CLK” and press Enter. The default port type is Avalon Memory Mapped Slave, but we want a clock input, click on the drop-down list for **Type**, and choose **Clock Input**.
   - In Quartus 16.0+, you are asked to choose from a list of port types, click on **Clock Input**, then rename it to “CLK” for **Name**.

   Finally, enter the **Clock rate** of 50MHz, your screen should look like this in both versions. If you added other interfaces by mistake, you can always right click those interfaces on the left tab and choose remove.



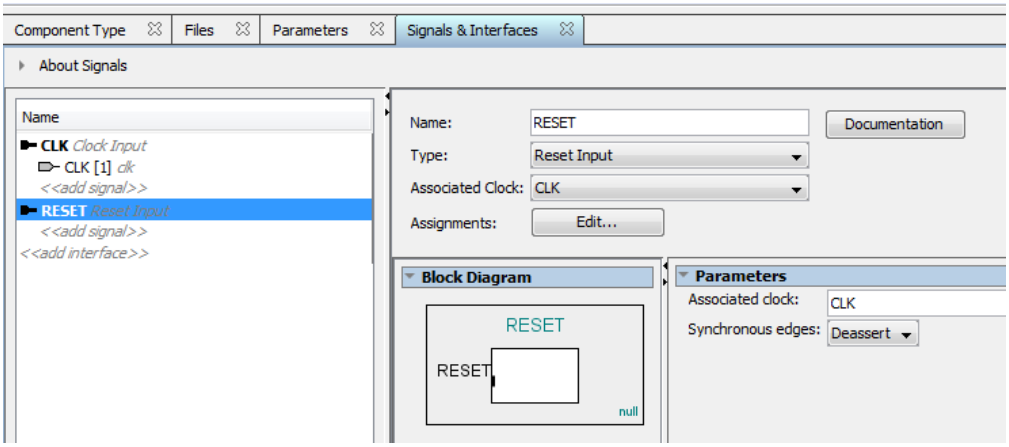
8. We now have a Clock Input interface for our component called CLK, but it's not associated with any signals defined in our top level module yet. Obviously, the clock input defined in *avalon_aes_interface.sv* is "input logic CLK". To make that association, click on *<<add signal>>*.
   - In Quartus 15.0, a default name of "new_signal" will appear, replace that with "CLK" to match the input name declared in *avalon_aes_interface.sv*.
   - In Quartus 16.0+, choose the only option of "clk", then change the Name on the right tab to "CLK" to match the input name declared in *avalon_aes_interface.sv*.

   The remaining default values should be correct, if not, change them to match the picture below.

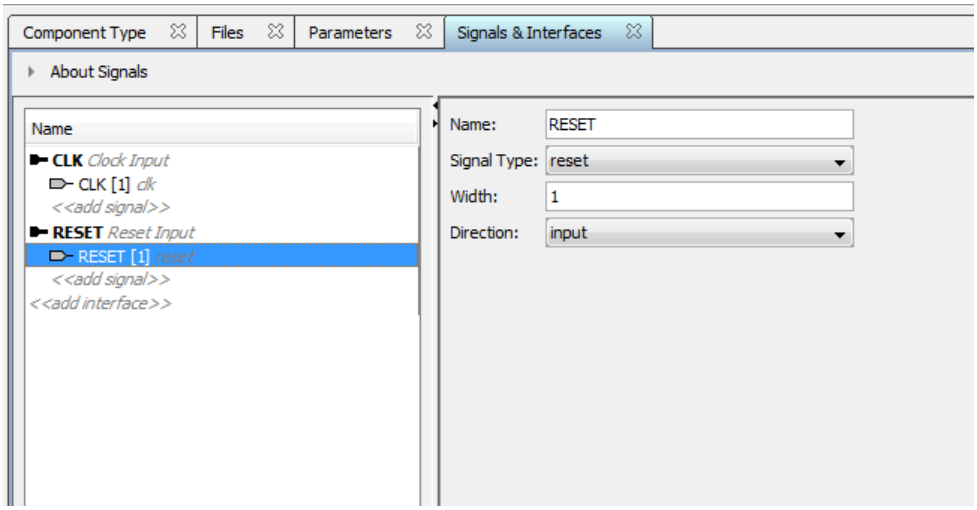


9. The Clock Input interface only needs one signal, so we are done with it. Now let's add the Reset Input interface. Click on *<<add interface>>*.
   - In Quartus 15.0, like the steps above, replace the default name with "RESET", then choose **Reset Input** for **Type**.
   - In Quartus 16.0+, likewise, choose **Reset Input** from the list of port types, then change the name to "RESET".

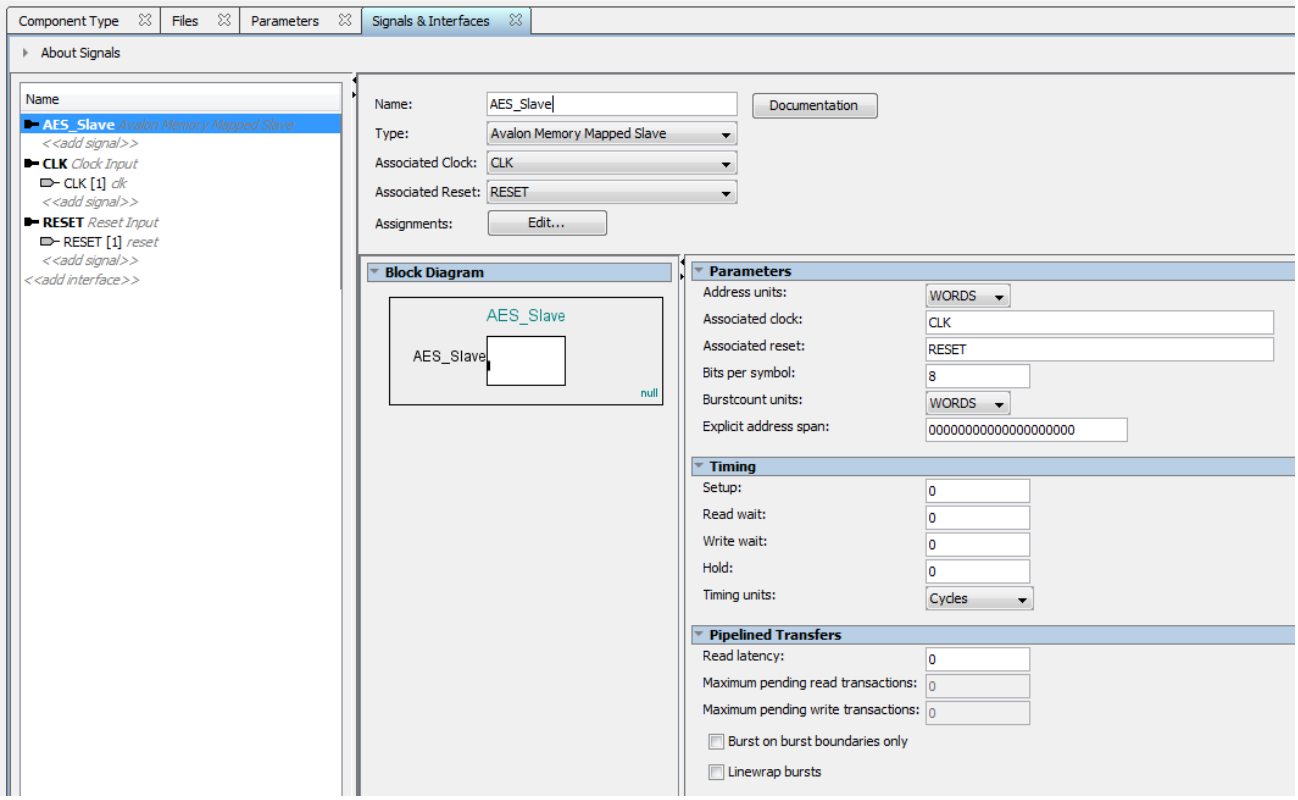
The associated clock should be set to the “CLK” interface we just defined by default, if not, set it.



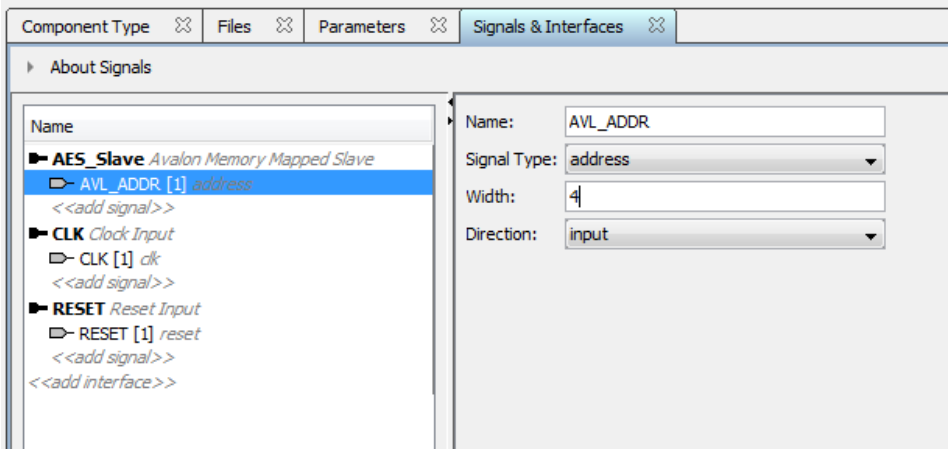
10. Now, add the associated RESET input to this interface. Click on <<*add signal*>>. Follow a similar procedure to Step 7, except use “RESET” for **Name**, and **reset** for **Type**.



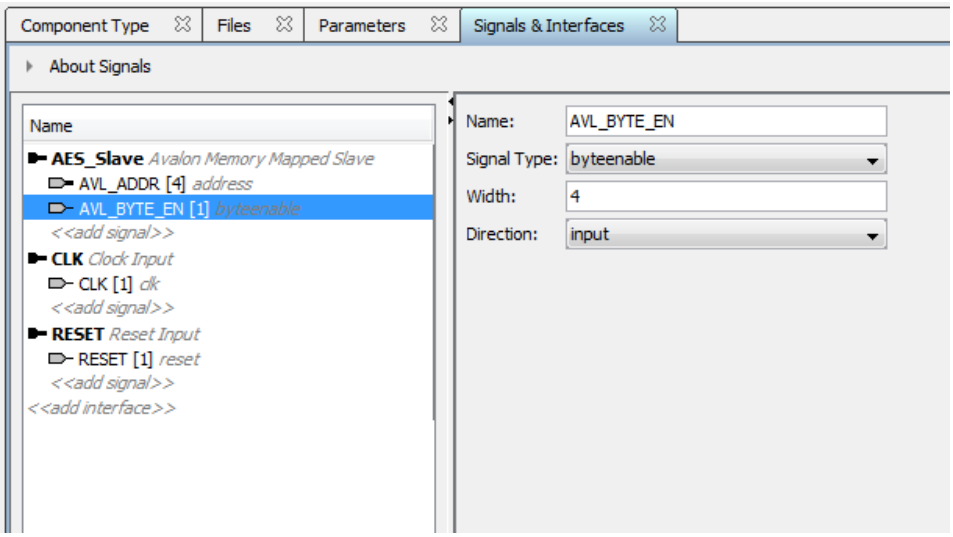
11. Next, add the main Avalon-MM Slave port. Click on <<*add interface*>>. You should be familiar with how to do this now, set the **Name** to “AES_Slave”, **Type** to **Avalon Memory Mapped Slave**. Also set the **Associated Clock** and **Reset** to the CLK and RESET ports we defined earlier.



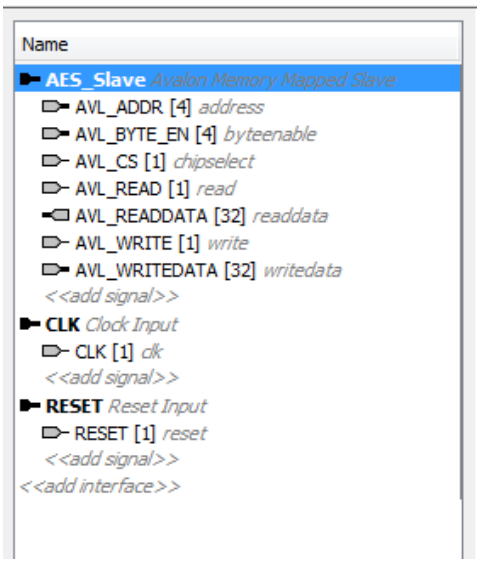
12. Under the **Parameters** section, check that you have identical settings as the picture above. Note that we make this Memory Mapped Slave byte addressable (8-bit per symbol) and the address units are in words (32-bit per word), so we expect the address to span a range of $4 \times 2^4 = 64$ (0x00 to 0x3F), making it no different than accessing no regular memory. In C, we can access this range directly as an array of 16 elements.
13. Under the **Timing** section, set both Read wait and Write wait to 0, as previously described, we want reads and writes to complete in the same cycle.
14. Now, we begin to add the relevant signals for this Memory Mapped slave port. Unlike previous ports, the MM Slave has multiple signals (read, write, chipselect, etc.) Let's start with the **address** signal, as before, click on <<*add signal*>> and set the **Name** to "AVL_ADDR", the **Signal Type** to **address**, **Width** to 4, and **Direction** to **input** in order to match what we declared in the top-level file *avalon_aes_interface.sv* ("input logic [3:0] AVL_ADDR").



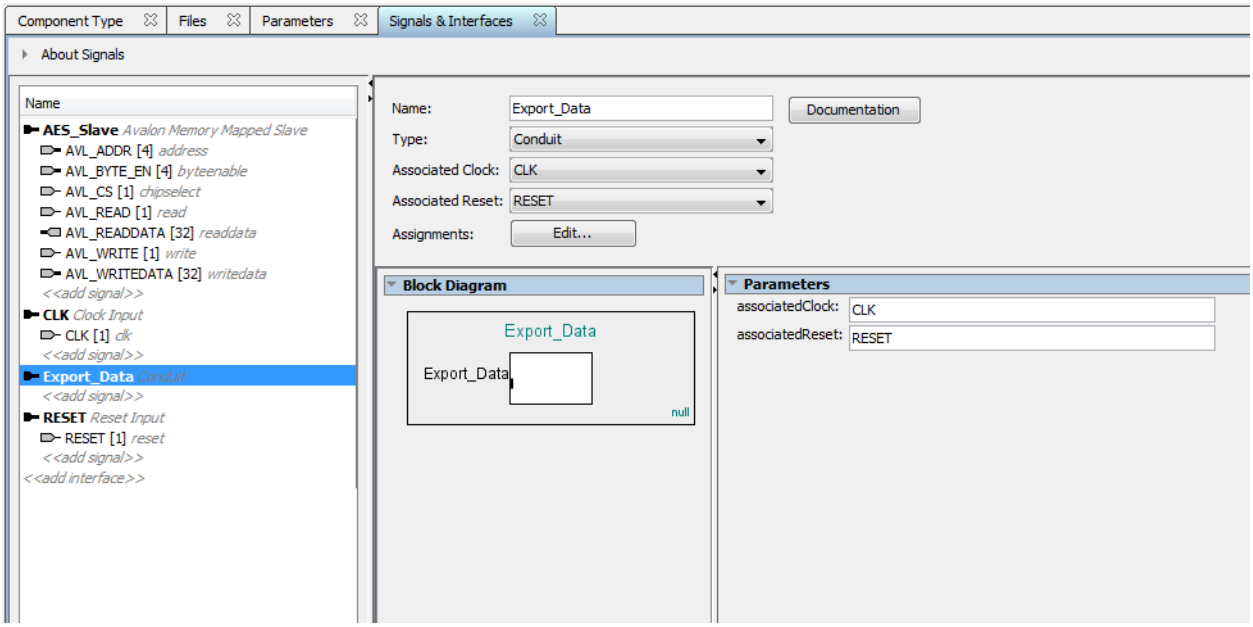
15. Add the **byteenable** signals next, click on <<*add signal*>> and set the **Name** to "AVL_BYTE_EN", the **Signal Type** to **byteenable**, **Width** to 4, and **Direction** to **input** once again to match what we declared in the top-level file *avalon_aes_interface.sv* ("input logic [3:0] AVL_BYTE_EN").



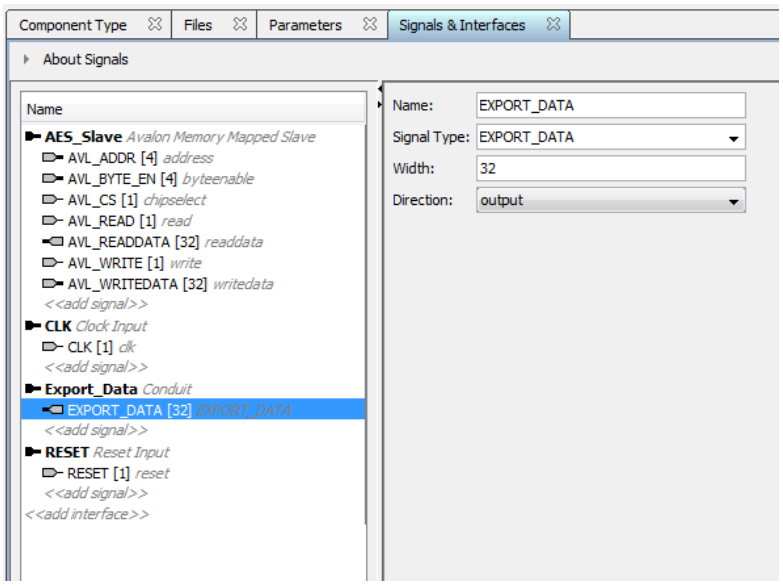
16. Complete the previous step the remaining signals for AES_Slave. Match the names, signal types, width, and direction for each signal. Be careful that **readdata** (AVL_READDATA) is an output unlike the others. When done, it should look like this:



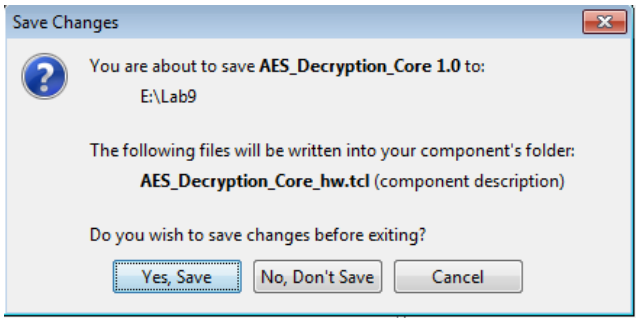
17. We are done with the AES_Slave. The last port to add is the exported conduit (EXPORT_DATA) to output part of the registers to the LEDs on the top level. Click on <<*add interface*>> and choose the settings as follows:



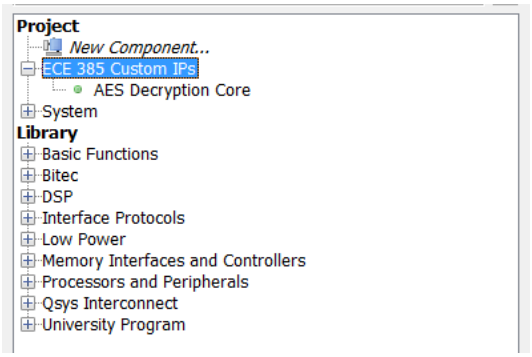
18. Add the only signal for this conduit, namely, the 32-bit EXPORT_DATA output signal. Set **Name** to EXPORT_DATA, **Signal Type** to * (it will be changed automatically or may appear as “new_signal” depending on your Quartus version), **Width** to 32, and **Direction** to output as shown below.



19. Go back to the Files tab, and click [Analyze Synthesis Files], you may get errors if you have bugs in your source files or if you instantiated other modules in avalon_aes_interface.sv, in the latter case, click [Add File…] to add them (e.g. AES.sv). Check that there are no more errors in the Message tab at the bottom and go back to the Signals & Interfaces tab to check that everything is still correct. If there are errors like "[*port name*] Interface must have an associate clock/reset", click on that port on the left tab, and set the Associated Clock or Reset to CLK and RESET respectively if they have been reset to none (this is a minor bug that happens in some versions of Quartus). If there are no errors, click on *Finish…* at the bottom right and click "Yes, Save" to save the generated TCL script.

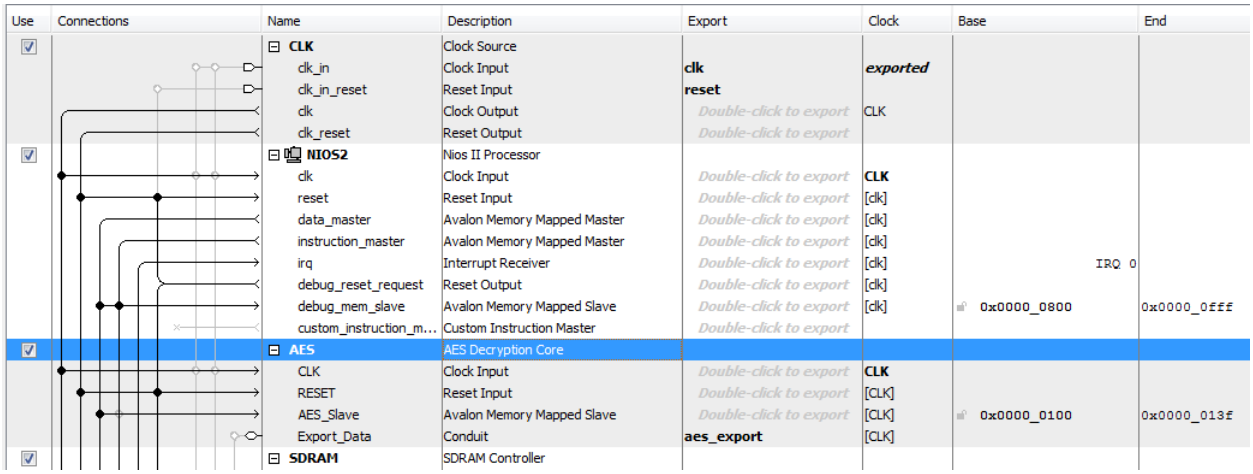


20. Now, in Platform Designer, your newly created AES Decryption Core component should appear in IP Catalog, grouped under "ECE 385 Custom IPs". It can now be added to Platform Designer like other Intel provided IPs. If you made a mistake, you can right-click it and click Edit to make any changes needed such as increasing the version number.



Finalizing your Platform Designer Design

Add your newly created component AES Decryption Core to Platform Designer by double clicking it. We did not define any parameters, so just click Finish to add it to Platform Designer, rename it if you want to. Make the appropriate **CLK** and **RESET** connections, and most importantly, connect the **AES_Slave** to Nios II's **data_master** to allow Nios to access its registers through reads and writes. We set its base address to 0x100 by default, you can choose a different address if it conflicts with your other components, and be sure to change it in software as well. Export the Export_Data conduit as "aes_export" and assign that signal to your HexDrivers on your top level file for this lab (*lab7_top.sv*).



Use	Connections	Name	Description	Export	Clock	Base	End
☑		**CLK**	Clock Source				
		clk_in	Clock Input	**clk**	***exported***		
		clk_in_reset	Reset Input	**reset**			
		clk	Clock Output	*Double-click to export*	CLK		
		clk_reset	Reset Output	*Double-click to export*			
☑		**NIOS2**	Nios II Processor				
		clk	Clock Input	*Double-click to export*	**CLK**		
		reset	Reset Input	*Double-click to export*	[clk]		
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]		
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]		
		irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	
		debug_reset_request	Reset Output	*Double-click to export*	[clk]		
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0800	0x0000_0fff
		custom_instruction_m...	Custom Instruction Master	*Double-click to export*			
☑		**AES**	AES Decryption Core				
		CLK	Clock Input	*Double-click to export*	**CLK**		
		RESET	Reset Input	*Double-click to export*	[CLK]		
		AES_Slave	Avalon Memory Mapped Slave	*Double-click to export*	[CLK]	0x0000_0100	0x0000_013f
		Export_Data	Conduit	**aes_export**	[CLK]		
☑		**SDRAM**	SDRAM Controller				

Finally, generate the HDL for your Platform Designer design and include the QIP in your project.

Recall that one of the main advantages of using hardware IPs is reusability. Using a standardized interface like Avalon allows your IP to be reused across different projects, even if other components change.

IMPORTANT: Whenever you change the SystemVerilog code in *avalon_aes_interface.sv* to fix bugs or add things after this, you need to use Platform Designer to regenerate the HDL so that those changes take effect. (The actual file being compiled by Quartus is the one generated by Platform Designer located in lab7\lab7_soc\synthesis\submodules\avalon_aes_interface.sv)

IMPORTANT: There is a bug in some versions of Quartus including 18.1 that causes it to misname your new Platform Designer component. If Platform Designer' generated top level Verilog file instantiates ***new_component*** for your ***avalon_aes_interface*** module, close Platform Designer, go to your project folder and edit the file AES_Decryption_Core_hw.tcl with a text editor: under the file sets section, manually change the TOP_LEVEL property to "avalon_aes_interface" as shown below, also check that the other lines are the same (unless your SystemVerilog file path is different). Then, open Platform Designer and update the version of your component (right click it under IP Catalog > Edit... and change version from 1.0 to 1.1 or any bigger number, then click Finish... to save), save your Platform Designer system and regenerate your HDL.

Alternatively, you can correct the instantiated module name in the Verilog file lab7_soc.v from "new_component" to "avalon_aes_interface". This needs to be done each time after you generate HDL in Platform Designer.

```
#
# file sets
#
add_fileset QUARTUS_SYNTH QUARTUS_SYNTH "" ""
set_fileset_property QUARTUS_SYNTH TOP_LEVEL avalon_aes_interface
set_fileset_property QUARTUS_SYNTH ENABLE_RELATIVE_INCLUDE_PATHS false
set_fileset_property QUARTUS_SYNTH ENABLE_FILE_OVERWRITE_MODE false
add_fileset_file avalon_aes_interface.sv SYSTEM_VERILOG PATH avalon_aes_interface.sv TOP_LEVEL_FILE

add_fileset SIM_VERILOG SIM_VERILOG "" ""
set_fileset_property SIM_VERILOG TOP_LEVEL avalon_aes_interface
set_fileset_property SIM_VERILOG ENABLE_RELATIVE_INCLUDE_PATHS false
set_fileset_property SIM_VERILOG ENABLE_FILE_OVERWRITE_MODE false
add_fileset_file avalon_aes_interface.sv SYSTEM_VERILOG PATH avalon_aes_interface.sv
```

Nios Software

Refer to how we defined the parameters for the Avalon-MM slave "AES_Slave" in step 12. To access its registers in a C program, we declare a pointer to the slave's base address.

```
// Pointer to base address of AES module, make sure it matches Platform
Designer

volatile unsigned int * AES_PTR = (unsigned int *) 0x00000100;
```

Since the Nios II/e processor has no cache or MMU, accessing the registers is as simple as dereferencing the AES_PTR pointer, for example:

```
// Send the 128-bit Key (Split into 4x 32-bit)

AES_PTR[0] = key[0];

AES_PTR[1] = key[1];

AES_PTR[2] = key[2];

AES_PTR[3] = key[3];
```

To test that your registers are working properly for week 1, you can write a value to it and read it back and check that it matches.

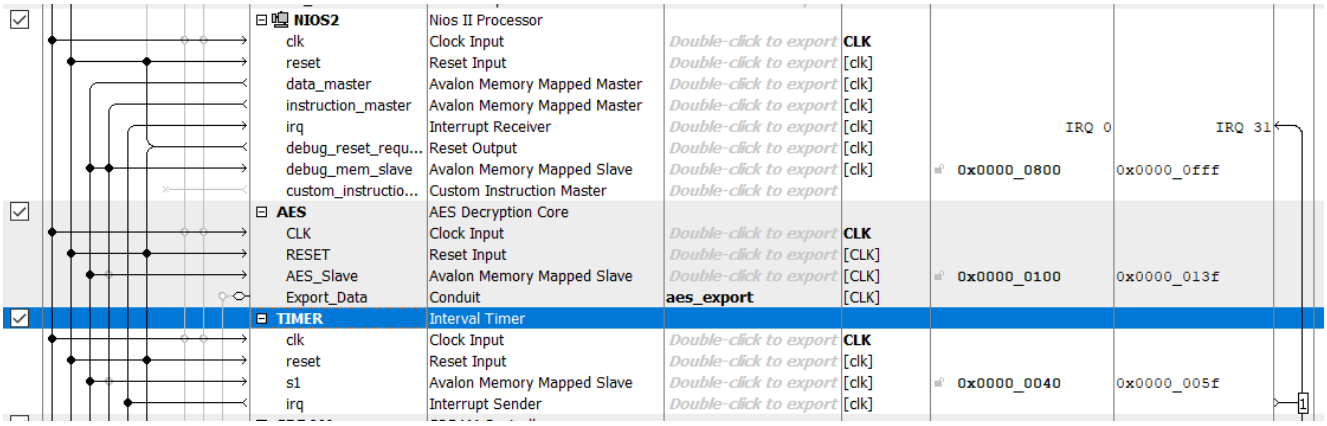
```
AES_PTR[10] = 0xDEADBEEF;

if (AES_PTR[10] != 0xDEADBEEF)

      printf("Error !");
```

For week 2, look at the state machine skeleton in Figure 3 to determine how you should control your hardware decryption module. Note that after sending the START signal (writing 1 to AES_PTR[14]), you should continuously read the DONE signal (AES_PTR[15]) until it becomes 1 which indicates that the decrypted message is ready, and finally, set the START signal to 0 to allow the state machine to return to WAIT state.

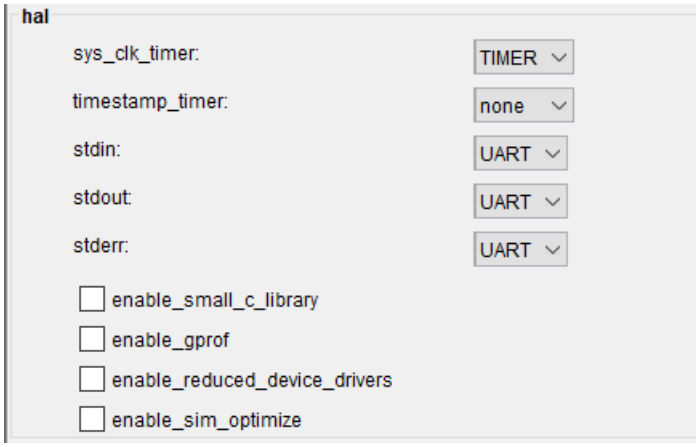
Nios II Benchmarking

When you are done testing the functionality of your code and hardware you can run your program in benchmark mode to get encryption/decryption speed measurements. However, Nios II itself does not have reliable means to measure time so we need to add an Interval Timer in Platform Designer that sends periodic interrupts.

1. Open your Platform Designer Project and add an Interval Timer. It can be found under "Processors and Peripherals" > "Peripherals" > Interval Timer. The default setting of 1ms intervals will suffice for our purposes.



2. Make the appropriate connections for each port, they should be familiar to you by now. Be sure to have the Interrupt Sender connected to Nios on the right side (in the example above, it is IRQ 1). As usual, you need to regenerate HDL and recompile your Quartus project.
3. In your Eclipse project, open BSP editor and choose your newly added interval timer for sys_clk_timer. This will make *time.h* function properly.



4. Generate BSP and recompile your Eclipse project before running.